V.07.

Общий обзор занятия.

Сегодня на разок отвлечёмся от программирования и займёмся организационными вопросами. Их много, так что мы только немного зацепим этот пласт. Но надо начинать. Поговорим о пакетах Go. Только о построении пакетов - и этого уже немало получается. Т.е. материала не так много, но его надо “уложить” и в голову, и в печёнку, сделать его “своим”, имплантировать его внутрь себя. Так что сегодня только построим пакеты и применим их в своём приложении.

Многие вопросы останутся пока “за кадром”. Какие? Например, мы не будем говорить о модулях Go (go.mod) - без этого пока удаётся проскользнуть, не будем даже затрагивать вопрос о публикации пакетов на github’е, как бы важно это ни было. Минимально и весьма формально введём новые возможности команды go, а именно, `go install` . Практически не будем говорить о переменных Go-среды - только упомянем про `$GOROOT` и пару особых каталогов: `$GOROOT\src` и `$GOROOT\bin` . Т.е. всё-таки говорить об организации среды Go будем совсем мало, в основном об организации Go-кода. Но надо начинать... Кроме довольно формального вброса в команды `go install` и `go build` можно показать детям команды `go env` , но это опционально. Неплохо бы показать `go help` и `go help <topic>` для самостоятельных штудий. Из ‘ов может быть есть смысл обратить внимание детей на build, env, возможно ещё какие-то, но увлекаться этим пока не стОит - надо дать материалу уложиться. Но залезать дальше и подробнее в инструментарий Go не будем - это отдельная и большая тема, и, пожалуй, не для этого сесместра - для следующего. В общем, как-то так.

Да, конечно говорим об экспортируемости сущностей (функций, переменных, типов, констант) из пакета. Ну, и до кучи к этому поговорим об областях видимости и блоках в Go - вопросы совсем небольшие, и говорить о них отдельно смысла нет, а здесь, вроде, получается уместным выделить для них уголок.

Лекция

Пакеты / Go Packages

Что такое пакеты и для чего они нужны

До сих пор все наши Go-программы состояли из одного файла с функцией main и парой-тройкой других функций. В реальном программировании так не работает - невозможно написать весь исходный код (исходный код - это как раз текст на языке программирования, на Go в нашем случае) в одном файле. Интеллигентно это называется не масштабируется. Далее, написанный таким образом код невозможно повторно использовать, практически невозможно сопровождать и поддерживать так написанный код. Вот тут-то и пригодятся пакеты.

Пакеты используются для организации исходного кода Go, позволяя использовать код (или его части) повторно, повышая читабельность кода. Пакет - это набор файлов с исходными текстами на Go, которые находятся в одном каталоге.

И ещё один очень важный момент, и это самое главное: **пакеты обеспечивают компартментализацию кода**. Этот хитрый термин надо, конечно, прояснить. Термин этот пришёл из биологии, и, грубо говоря, означает разделение клеток на отсеки (компартменты), в которых локализованы определенные биохимические процессы. Собственно, compartment дословно означает отсек. Т.е. компартмент - это своего рода орган, часть, чётко, физически отделённая от других частей и выполняющая какие-то определённые функции. Принцип компартментализации позволяет клетке выполнять разные процессы одновременно.

В программировании ситуация схожая. Учитывая, что программа - это живой организм (а я лично в этом совершенно убеждён - программы точно также, как и живые организмы, рождаются, развиваются, проходят различные стадии своего развития, и точно также перестают развиваться - умирают), то компартментализация удобно разделяет проект на отсеки - на части, каждая из которых, во-первых, имеет свою связную функциональность, а во-вторых,

позволяет чётко, физически отделить эти части одну от другой. Это существенно облегчает поддержку проектов Go. Можно достаточно писать части проекта раздельно, не задумываясь о судьбе других частей, беспокоясь только об интерфейсах с ними, а интерфейсы между частями (компартментами), конечно, оговариваются заранее и всё время поддерживаются.

Кратко основные идеи: пакеты Go - это обычный Go-код. Однако размещение кода пакетов в собственных каталогах (директориях) позволяет изолировать код, именно физически изолировать. В результате этим кодом можно воспользоваться в любых других Go-проектах.

Поговорим, конечно, о техническом моменте: как объявлять в пакетах функции, переменные и типы, как их экспортировать - организовать, обеспечить использование этих сущностей за пределами пакетов, где следует хранить пакеты, рассчитанные на многократное использование и т.д.

Описание примера. План действий.

Мы рассмотрим пример, в котором мы будем выполнять операции, связанные с целочисленным возведением в степень, а именно:

- a^N - N-я степень числа a,
- целый корень N-й степени из числа x - т.е. наибольшее целое число a, N-я степень которого не превосходит x,
- целый логарифм числа x по основанию a - наибольшая целая степень N, такая что a^N не превосходит x.

Да, сюжет математизирован и носит несколько абстрактный характер, но этакая абстактноватость вполне уместна для повторного использования кода. А то замучили всякие финансовые функции-применения, которыми полна сеть. Да и разбираться с ними потяжелее, чем с такой вот просто арифметикой.

При этом для разбега будем применять самые тупые алгоритмы, чтобы не заморачиваться. А впоследствии можем запросто изменить реализации функций на более эффективные, и это никак не скажется на приложениях, которые их используют - достаточно будет всё просто перекомпилировать.

Создадим для того аж целых три пакета - power, root и logarithm. Точнее говоря, мы

напишем три файла с исходным go-текстом, пакетов создадим два - power и aroundpower, в котором мы объединим код из файлов root.go и logarithm.go. При реализации пакета aroundpower будем вычислять степень, а для этого использовать пакет power - вот вам и повторное использование кода.

Начнём с простого приложения, вычисляющего степень числа.

Внимание. По ходу пьесы мы будем постепенно наращивать файлы исходных текстов. Конечный результат будет такой:

```
learnpackage
|   app.go
|
+---power
|       power.go
|
+---aroundpower
|       logarithm.go
|       root.go
|
\---temporarilyUnusedPackage
        unused.go
```

Все промежуточные файлы мы будем хранить навалом в каталоге `tmp`, сопровожая их названия номерами версий, но при этом в тексте плана я буду использовать их “настоящие”, окончательные названия. Так что, показывая что-то, временные файлы надо будет или переименовываать, или вносить соответствующие изменения в команды .

main function и main package

Каждое исполняемое Go-приложение должно содержать функцию main. Эта функция и есть точка входа, точка начала исполнения приложения. Функция main должна находиться в пакете main.

Первая строчка любого файла с исходным go-текстом должна начинаться со строки

```
package packagename
```

Это указывает на то, что конкретный исходный файл принадлежит пакету packagename.

Создаём для начала функцию main function и пакет main для нашего приложения. Создаём каталог (директорию, папку)

```
md learnpackage
```

В этом каталоге создаём файл app.go (версия app0.go) с таким содержанием :

```
package main

import "fmt"

func main() {
    fmt.Println("Power calculation")
}
```

Первая строчка, повторю, означает, что файл app.go принадлежит пакету main. Строка `import fmt` используется для импортирования из имеющегося пакета (fmt в данном случае) и функций (в данном случае функции fmt.Println()) и других сущностей (типов, констант, переменных).

fmt - это стандартный пакет, он доступен как встроенная часть стандартной библиотеки Go.

Компилируем полученный осходный код. Привычная нам компиляция из какой-либо среды программирования сегодня нас не спасёт, так что сразу будем компилировать из командной строки. Для этого перейдём в каталог learnpackage и дадим в командной строке соответствующую команду go build:

```
go build app.go
```

В результате получаем исполняемый файл app.exe, который работает вполне себе ожидаемо - печатает строку `Power calculation` .

Но давайте теперь поступим совсем по Go-шному. Каталог `learnpackage` вместе с исходным файлом app.go поместим в каталог `C:\Go\src`, зайдём в него (это всё делаем в командной строке с помощью команды `cd`) и дадим команду

```
go install
```

Эта команда найдёт в нашем каталоге package main, откомпилирует его и результат поместит в каталог `C:\Go\bin`.
Выполняемый файл получит название `learnpackage.exe`.

Мы будем структурировать код таким образом, чтобы все функциональные возможности, связанные с power, находились в пакете power. Для этого нам нужно создать пользовательский пакет (custom package) power, который будет содержать функцию для вычисления степени.

Создаём пользовательский пакет power

Исходные файлы, принадлежащие пакету, должны быть помещены в отдельные каталоги. В Go принято соглашение называть этот каталог тем же именем, что и пакет.
Создаём каталог power внутри каталога `learnpackage`. Напомнию, что это делает команда md, она же mkdir:

```
md power
```

Все файлы внутри каталога power должны начинаться со строки

```
package power
```

поскольку они все принадлежат пакету power.

Внутри каталога power создаём файл power.go (версия power0.go):

```
//power.go
package power

func Power(a int64, n uint) int64 {
```

```
    var (
        i uint
        res int64
    )
    for i, res = 0, 1; i < n; i++ {
        res *= a
    }
    return res
}
```

Обратите внимание, что название функции Power начинается с большой буквы. Это существенно, но об этом чуть позже.

Подаём для контроля команду (ещё раз - из командной строки!)

```
C:\Go\src>tree learnpackage /F
```

и получим:

```
C:\GO\SRC\LEARNPACKAGE
├── learnpackage
│       app.go
│
└─────power
            power.go
```

Импортируем пользовательский пакет

Чтобы использовать пользовательский пакет, мы должны его импортировать его в приложение. При этом надо указывать весь путь для импортирования. В нашем случае имя модуля learnpackage, а пакет power находится в каталоге learnpackage\power. Пакет fmt входит в стандартную библиотеку Go, поэтому путь к нему указывать не надо.

Импортируем пакет power в app.go (версия app1.go) и применяем его:

```
package main
```

```
import  (
    "fmt"
    "learnpackage/power"
)

func main() {
    var (
        a int64 = 3
        n uint = 5
    )
    p:= power.Power(a, n)
    fmt.Println(a, "^", n, "=", p)
}
```

Заходим в каталог `C:\Go\src\learnpackage` и компилируем полученный код. Можем, конечно для этого дать команду

```
go build app.go
```

и тогда мы получим приложение `C:\Go\src\learnpackage\app.exe`, но давайте дадим команду

```
go install
```

В результате получим исполняемый файл `C:\Go\bin\learnpackage.exe`.

Запускаем его и видим результат:

```
Power calculation
3 ^ 5 = 243
```

Чуть подробнее о команде go install

Инструменты Go типа go install работают в контексте текущего каталога. До сих пор мы запускали `go install` из каталога `C:\Go\src\learnpackage`. Попытка запустить

эту команду из другого каталога завершится обломом. Вот, например, запустим `go install` из каталога `C:\Go\src\learnpackage`:

```
C:\Go\src>go install
```

и увидим

```
can't load package: package .: no Go files in C:\Go\src
```

Ну, ничего. Попробуем так:

```
C:\Go\src>go install learnpackage
```

Всё ok, в каталоге `C:\Go\bin` появляется файл `learnpackage.exe`

Но попробуем двинуться чуть дальше, и мы с удивлением и радостью поймём, что эта команда работает точно также, будучи запущенной из любого каталога, хоть даже и с другого диска.

В чём же дело? Давайте разбираться. Команда `go install` должна запускаться с параметром - имя пакета. Однако этот параметр опционален - его можно не указывать. В таком случае команда `go install` считает именем пакета имя текущего рабочего каталога. Поэтому `go install` и работает, если её запустить из каталога `C:\Go\src\learnpackage`, и не работает из других каталогов.

А почему каталог learnpackage ищется именно в каталоге `C:\Go\src`? На самом деле это не обязательно, для этого существует специальное понятие модуль Go и файл модуля `go.mod`. Но давайте мы пока такую возможность проигнорируем, у нас сегодня и без этого много других задач. Так что, все наши исходные файлы импортируемых пакетов, собранные в каталог learnpackage (совпадает с именем пакета), поместим именно в каталог src (это фиксированное имя, с ним мы бороться не будем), который лежит именно в каталоге
'C:\Go\'. А вот этот каталог есть значение переменной Go-среды (Go environment variable) GOROOT. Её можно изменить, но об этом тоже не сегодня...
Да, и результат работы `go install` помещается в `C:\Go\bin` по тем же причинам: значение переменной Go-среды GOROOT плюс bin - фиксированное

название.

И всё-таки, ещё про сборку exe-файла с помощью go build

Можно всё-таки поступить чуть проще. А именно. Каталоги и исходные файлы пакетов помещаем в `C:\Go\src\learnpackage`. А вот файл app.go с package main держим в любом другом каталоге, и, находясь в этом каталоге, подадим команду `go build app.go`. В результате получим выполняемый файл `app.exe`.

Экспортируемые имена (Exported names)

Мы назвали функцию Power в пакете power с большой буквы, и это, как уже говорилось, имеет особый смысл. Почему? А потому, что любой пакет экспортирует только те сущности (типы, функции, переменные, константы и т.д.), которые названы с большой буквы. И только экспортируемые сущности доступны из других пакетов, могут быть импортированы в них.

Добавим в power.go (версия power1.go) пару коснтант:

```
const (
  Two = 2
  three = 3
)
```

и изменим пакет main, который живёт в файле app.go (версия app2.go), таким образом:

```
package main

import  (
    "fmt"
    "learnpackage/power"
)

func main() {
```

```
    fmt.Println("Power calculation")
    var (
        a int64 = 3
        n uint = 5
    )
    p:= power.Power(a, n)
    fmt.Println(a, "^", n, "=", p)
    fmt.Println(power.Power(power.Two, 5))
    fmt.Println(power.Power(power.Three, 2))
    fmt.Println(power.Power(power.three, 2))
}
```

компилируем

```
F:\>go install learnpackage
```

и получаем

```
# learnpackage
c:\go\src\learnpackage\app.go:17:29: undefined: power.Three
c:\go\src\learnpackage\app.go:18:29: cannot refer to unexported name power.three
c:\go\src\learnpackage\app.go:18:29: undefined: power.three
```

И это вовсе не потому, что запустили команду с диска F: - я умышленно запустил с другого диска, чтобы проиллюстрировать ещё раз вышесказангное про $GOROOT. А почему именно, ясно написано: power.Three вовсе не определена, а имя power.three хоть и определено в пакете power, но оно не экспортируемое, соответственно в package main (в файле app.go) имя power.three не определено.

Зачем нам вообще что-то запрещать к экспорту? Ну, это совершенно естественно - чтобы как-то локализовать внутри пакета то, что не предусмотрено к экспортированию, то, что задумано как чисто внутреннее дело пакета, что используется исключительно с целью реализации (имплементации) каких-то других действий в этом пакете, то что носит локальный характер. Лишнее - мешает, так что не надо экспортировать то, что нужно только для внутренних нужд. Это в определённом смысле что-то наподобие локальных переменных внутри функции, что-то вроде области видимости для пакета. Об областях

видимости - чуть попозже. Неэкспортируемые сущности не видны извне пакета.

функция init

Идентификатор (имя) init имеет особый смысл и может использоваться только для функции инициализации пакета. Каждый пакет может содержать функцию init:

- Функция init не может быть вызвана явно в нашем исходном коде. Она вызывается автоматически при инициализации пакета, при старте.
- Функция init имеет следующий синтаксис:

```
func init() {
  // function body
}
```

- Функция init не должна иметь никаких параметров и не должна ничего возвращать.
- Функция init используется для выполнения задач инициализации, а также для проверки правильности работы программы перед началом выполнения.

Порядок инициализации пакета следующий:

1. Первым делом инициализируются переменные уровня пакета. Переменные уровня пакета - это те переменные, которые объявлены вне функций, подробнее об этом - ниже.
2. А затем вызывается функция init.
    - Пакет может иметь несколько функций init (либо в одном файле, либо распределенных по нескольким файлам), и они вызываются в том порядке, в котором они представлены компилятору.
    - Если пакет импортирует другие пакеты, то импортированные пакеты инициализируются первыми.
3. Пакет будет инициализирован только один раз, даже если он импортирован из нескольких пакетов.

Посмотрим всё это на примере:

Пример

В оба оба файла, составляющих `package aroundpower` вставим сразу функции init(). И это будут сразу окончательные версии этих файлов - root.go и logarithm.go:

```go
//root.go
package aroundpower

import (
    "fmt"
    "learnpackage/power"
)

func init()  {
   fmt.Println("aroundpower package initializing... - root")
}

func Root(a int64, n uint) int64 {
    var i int64
    for i = 0; power.Power(i, n) <= a; i++ {  }
    return i-1
}
```

```go
//logarithm.go
package aroundpower

import  (
    "fmt"
    _ "learnpackage/power"
)

func init()  {
   fmt.Println("learnpackage/aroundpower package initializing... - logarithm")
}

func Logarithm(a int64, base uint) int64 {
    if base <= 1 || a <= 0 { return -1 }
    var  (
        res int64 = 0
        p int64 = 1
    )
```

```
    for p <= a  {
        p *= int64(base)
        res++
    }
    return res-1
}
```

В `logarithm.go` имеется странная строчка

```
    _ "learnpackage/power"
```

в которой вроде бы импортируется пакет “learnpackage/power”, но как-то странно импортируется, с пустым идентификатором `_` перед ним. Да и не нужен этот пакет, вроде. Да, не нужен, да импортируется, да компилятор ошибки не нахоит, но об этом чуть-чуть ниже. А пока я привёл весь текст, не вставлять же его дважды из-за этой одной строчки, которая к тому же разъяснится уже следующим шагом.

А ещё добавим функцию init() в package main, причём там же ещё и объявим на уровне пакета переменную title - файл app3.go:

```
package main

import  (
    "fmt"
    "learnpackage/power"
    "learnpackage/aroundpower"
)

var title = "main.init() function processing..."

func init() {
  fmt.Println(title)
}

func main() {
    var (
```

```
        a int64 = 3
        n uint = 5
    )
    p:= power.Power(a, n)
    fmt.Println(a, "^", n, "=", p)
    fmt.Println(power.Two, "^ 5 =", power.Power(power.Two, 5))
    fmt.Println("cubic root of 250 = ", aroundpower.Root(250, 3))
    // 6^3 = 216 < 250 < 7^3 = 343
    fmt.Println("fifth degree root of 50000 = ", aroundpower.Root(50000, 5))
    // 8^5 = 32768 < 50000 < 9^5 = 59049
    fmt.Print("the logarithm of 250 to base 3 = ")
    fmt.Println(aroundpower.Logarithm(250, 3))
    // 3^5 = 243 < 250 < 3^6 = 729
}
```

компилируем командой `go install learnpackage`, запускаем и получаем вот что:

```
aroundpower package initializing... - logarithm
aroundpower package initializing... - root
main.init() function processing...
3 ^ 5 = 243
2 ^ 5 = 32
cubic root of 250 =  6
fifth degree root of 50000 =  8
the logarithm of 250 to base 3 = 5
```

как видно, порядок инициализации такой:

- первым делом инициализируются импортируемые пакеты (точнее, конечно, один пакет - aroundpower); при этом запускаются поочерёдно обе функции init() - и в файле logarithm.go, и в файле power.go
- затем начинается инициализиация пакета main: сначала инициализируется переменная title уровня пакета, а затем уже вызывается функция init() этого пакета
- и, наконец, вызывается функция main()

Как бороться с возникающей ошибкой компиляции

при неиспользовании импортированного пакета - использование пустого (blank) идентификатора `_`

Да, в Go запрещено импортировать пакет и при этом не использовать его. Кадется, что это вполне естественно, в самом деле, зачем импортировать что-то, совсем даже ненужное, ведь лишнее - мешает, да и время компиляции увеличивается. Но с появлением понимания процесса инициализации пакетов ситуация изменяется: вполне себе возможно, что мы заинтересованы в инициализации какого-то пакета, даже если мы напрямую не используем никаких сущностей из него. Да хоты бы и потому, что мы планируем в дальнейшем его использовать, а сейчас, в процессе активной разработки приложения, этот пакет пока не используется.

И в такой ситуации нам поможет заглушить ошибку пустой идентификатор.

```
package main

import (
    _ "learnpackage/temporarilyUnusedPackage"
)

func main() {

}
```

Проверим, как это работает. Создадим каталог `learnpackage\temporarilyUnusedPackage`, а в нём файл `unused.go` (сразу уже в окончательной версии):

```
//unused.go
package temporarilyUnusedPackage

import "fmt"

func init() {
    fmt.Println("learnpackage/temporarilyUnusedPackage initialized")
```

```
}
```

импортируем в app.go этот пакет (наконец-то окончательный вариант app.go), откомпилируем и запустим приложение. Получим

```
learnpackage/aroundpower package initializing... - logarithm
learnpackage/aroundpower package initializing... - root
learnpackage/temporarilyUnusedPackage initialized
main.init() function processing...
3 ^ 5 = 243
2 ^ 5 = 32
cubic root of 250 =  6
fifth degree root of 50000 =  8
the logarithm of 250 to base 3 = 5
```

Как видно, пакет `learnpackage/temporarilyUnusedPackage` проинициализировался вполне себе успешно.

Да, а ещё до кучи мы импортировали в файл `learnpackage\logarithm.go` не используемый в нём пакет “learnpackage/power”:

```
//logarithm.go
package aroundpower

import  (
    "fmt"
    _ "learnpackage/power"
)
...
```

и убедились, что этот пакет (`learnpackage/power`) повторно не инициализируется, хотя и импортирован трижды - из `app.go` , из `learnpackage\logarithm.go` и из `learnpackage\root.go` .

Пожалуй, хватит о пакетах на сегодня. Но до кучи давайте быстренько поговорим об областях видимости (scopes) в Go.

Области видимости (scopes) и блоки

Область видимости (можно сказать область действия, хотя так говорят редко) идентификатора - это часть исходного кода (возможно, весь код), где идентификатор привязывается к определенной сущности, такой как переменная, константа, тип, пакет и т.д.

Определение области видимости жёстко связано с местом, где объявлен идентификатор). Областью видимости идентификатора является **блок**.

Блок в исходном тексте Go - это несколько последовательных строк кода. Блоки явно выделяются фигурными скобками `{` и `}`. Блоки могут быть вложенными один в другой. Есть ещё неявные блоки, но о них чуть ниже, сначала разберёмся с явно объявленными блоками и соответствующими областями видимости.

Область видимости идентификатора переменной или константы охватывает самый внутренний содержащий её блок (явный, обозначенный фигурными скобками, либо неявный).

пример scopes01.go.

```
package main

import "fmt"

func main() {
    {
        v := 1
        {
            fmt.Println(v)        // 1
        }
        fmt.Println(v)
    }
    // fmt.Println(v)
    //             compilation failed - undefined v
}
```

пример scopes02.go.

```
package main

import "fmt"

func main() {
    { // start outer block
        a := 1
        fmt.Println(a)         // 1

        { // start inner block
            b := 2
            fmt.Println(a, b)  // 1 2
        } // end inner block
        fmt.Println(a)         // 1
        // fmt.Println(b)       - undefined: b

    } // end outer block
    // fmt.Println(a)       - undefined: a
}
```

Заметим, что для переменных и констант область их видимости начинается после окончания всего оператора объявления. Так что писать вот так:

```
    a, b := 1, a
```

нельзя - переменная a в правой части присваивания ещё не видима.

В то же время для типов область видимости точно также охватывает весь самый внутренний блок, в котором тип объявлен, но начинается не в конце объявления, а в конце написания идентификатора. Это дополнительное пространство видимости позволяет определять рекурсивные типы, а это важно для объявления структуробразующих типов, мы это видели буквально на последних двух-трёх занятиях.
пример scopes03.go.

```
package main

import "fmt"

func main() {
    type line struct {
        name string
        next *line
    }

    x := line{name: "Max", next: &line{name: "Alex", next: nil}}
    fmt.Println(x.name)           // Max
    fmt.Println(x.next.name)      // Alex
    fmt.Println(x.next.next)      // <nil>
}
```

Неявные блоки и соответствующие области видимости

Кроме явно помеченных (явных) блоков, есть еще несколько неявных:

- всеобщий блок (universe block) содержит весь исходный код,
- блок пакета (package block) содержит весь исходный код пакета; напомню, пакет может быть распределен по нескольким файлам в одном каталоге,
- блок файла (file block) содержит исходный код файла,
- предложение for неявно входит в свой явно отмеченный блок:
  package main

пример scopes04.go.

```
import "fmt"

func main() {

    for i := 1; i <= 5; i++ {
        fmt.Print(i)          // 12345
    }
```

```
    fmt.Println()
    //  fmt.Println(i)    - undefined: i

    var i int                   // no redeclaring
    for i = 1; i <= 5; i++ {
        fmt.Print(i)            // 12345
    }
    /// var i int          -    i redeclared in this block
    fmt.Println()
    fmt.Println(i)              // 6
}
```

- совершенно аналогично предложение if неявно входит в свой явно отмеченный блок. Всё совпадает с for, так что я не буду писать полного примера, ограничусь чем-то вроде такого:

```
    if i := 0; i >= 0 {
        fmt.Println(i)
    }
```

- и абсолютно точно также в свой явно отмеченный блок входит предложение switch:
  пример scopes05.go.

```
package main

import "fmt"

func main() {
    switch i := 10; i % 2 {
    case 0:
        fmt.Println(i, "is even integer")
    case 1:
        fmt.Println(i, "is odd integer")
    default:
        fmt.Println("Wow! Amazing", i)
    }
```

```
}
```

- каждый вариант (предложениее) в операторе switch действует как неявный блок, т.е. запись

```
    switch i := 2; i * 4 {
    case 8:
        j := 0
        fmt.Println(i, j)
    default:
        // j is undefined here
        fmt.Println(“default”)
    }
    // j is undefined here, as well as i
```

эквивалентна записи

```
    switch i := 2; i * 4 {
    case 8:  {
        j := 0
        fmt.Println(i, j)
    }
    default:  {
        // j is undefined here
        fmt.Println(“default”)
    }
    }
    // j is undefined here, as well as i
```

но фигурные скобки, ограничивающие такие неявные блоки можно (и принято) не указывать явно для сокращения и повышения читабельности кода.

- и совершеннот точно также каждый варинат в предложении `select statement` действует как неявный блок:

```
    tick := time.Tick(100 * time.Millisecond)
LOOP:
```

```
    for {
        select {
        case <-tick:
            i := 0
            fmt.Println("tick", i)
            break LOOP
        default:
            // "i" is undefined here
            fmt.Println("sleep")
            time.Sleep(30 * time.Millisecond)
        }
    }
    // "i" is undefined here
```

Предопределённые идентификаторы

Имеется набор встроенных идентификаторов:

- типы: bool, int32, int64, float64, ...
- nil
- функции: make, new, panic, ...
- константы вроде true или false

Все они имеют областью действия всеобщий блок, поэтому доступны везде.

Импорт

При импорте пакетов область действия имён - файловый блок. Так что если пакет состоит из нескольких файлов, то при импорте чего-то в одном файле этого пакета, это импортирование не действует в другом файла этого же пакета.

Идентификаторы на верхнем уровне

Переменные, константы, типы, функции, объявленные вне какой-либо любой функции, видны во всем пакете, во всех файлах пакета.

Функции и методы

Параметры функции или переменная результата и ресиверы методов видны только в теле функции.

Зачем нужны области видимости?

Они дают возможность повторно использовать одно и то же название для разных переменных, для переменных с различным содержанием в различных частях кода. Для больших программ это весьма полезно - пропадает необходимость запоминать все переменные. Как только переменная выходит из области видимости, можно больше не думать о ней.

Практика.

Приведу краткое описание нескольких пакетов, которые можно написать.

1. Работа с рациональными числами. Каждое число хранится как дробь, в которой числитель и знаменатель есть целые числа. Хорошо бы завести

```
type Fraction struct  {
    a int  // numerator
    b int  // denominator
}
```

и к нему несколько методов. Например,

```
func (x *Fraction) Inc (y Fraction)
    x <- x + y
func (x *Fraction) Dec (y Fraction)
    x <- x - y
func (x *Fraction) Mult (y Fraction)
    x <- x * y
func (x *Fraction) Div (y Fraction)
    x <- x / y
func (x Fraction) Compare (y Fraction) int
    -1, если x < y,
     0, если x == y,
```

```
     1, если x > y,
func (x *Fraction) Reduce()
    Сокращает дробь. Тут как раз появляется возможность
    написать неэкспортруемую функцию для поиска
    наибольшего общего делителя числителя и знаменателя.
func (x Fraction) Round() int
   Округляет дробь до ближайшего целого
func (x Fraction) Floor() int
   Округляет дробь вниз до целого
func (x Fraction) Ceil() int
   Округляет дробь вверх до целого
func (x Fraction) Frac() Fraction
   Дробная часть x
```

и т.д.

2. Пакет для работы с длинной арифметикой. Числа (целые числа) можно хранить, например, как слайс цифр. При этом, думаю, будет удобно хранить цифры от конца к началу, т.е. в нулевом элементе слайса хранить единицы, в первом - десятки, во втором - сотни и т.д. Это позволяет легче ориентироваться со степенями десятки - индекс элемента слайса совпадает с соответствующей степенью 10.

И тут тоже хорошо бы завести соответствующий тип и написать к нему методы. Расписывать методы я тут не стану, тут и так всё понятно, да и многие методы аналогичны перечисленным в первом примере.